An Overview of Machine Learning and Deep Learning

Why Mathematics Matters in Deep Learning

Before diving into the technical content, let's address a fundamental question: **Why study the mathematics of machine learning?**

The internet provides abundant prebuilt deep learning models and training systems that require minimal understanding of underlying principles. However, practical problems often do not fit any publicly available models. These situations demand **custom model architecture development**, which requires understanding the mathematical underpinnings of optimization and machine learning.

Key insight: Programming skills (particularly Python) are mandatory, but without intuitive mathematical understanding, the *how* and *why* behind model decisions—and crucially, the answer to "Can I repurpose this model?"—will remain opaque. Mathematics allows you to see the **abstractions behind the implementation**.

> ***Recommended practice:*** *For maximum benefit, work out the mathematics with paper and pencil, and execute code implementations on a computer.*

---

1. Machine Learning as Function Approximation

The Core Paradigm Shift

Machine learning represents a fundamentally different paradigm of computing:

Traditional Computing	Machine Learning
Provide step-by-step instructions (program)	Build a mathematical model
Explicitly define input → output rules	Learn the transformation from data

Programmer encodes the solution	Model discovers the solution

Central idea: Machine learning is nothing but **function approximation**—we are simply trying to approximate the unknown classification or estimation function.

In typical real-life situations, we do not know the true transformation function. For instance, we don't know the function that takes past prices, world events, etc. and estimates future stock prices. All we have is **training data**—a set of inputs where the output is known.

---

2. The Basic Machine Learning Model

The Cat Brain Example

Consider a hypothetical cat brain that must make one decision: whether to **run away**, **ignore**, or **approach and purr** at an object. This decision is based on two quantitative inputs (features):

- **x_0**: perceived hardness
- **x_1**: perceived sharpness

The Linear Model

The simplest model architecture is a **weighted sum of inputs**:

$$y(\text{hardness}, \text{sharpness}) = w_0 \times \text{hardness} + w_1 \times \text{sharpness} + b$$

Or in vector notation:

$$y(x_0, x_1) = w_0 x_0 + w_1 x_1 + b = \vec{w}^T \vec{x} + b$$

Terminology:

- **Weights** (w_0, w_1): Parameters multiplied by inputs
- **Bias** (b): Constant term not multiplied by any input

- **Output** (y): Interpreted as a "threat score"

Decision rule (Equation 1.2):

$$y \begin{cases} > \tau & \rightarrow \text{high threat, run away} \\ \geq -\tau \text{ and } \leq \tau & \rightarrow \text{threat close to zero, ignore} \\ < -\tau & \rightarrow \text{negative threat, approach and purr} \end{cases}$$

---

3. The Machine Learning Pipeline

Three Stages of Problem Solving

Stage 1: Model Architecture Selection

- Design a parameterized model function with unknown parameters (weights)
- This is where the expertise of the ML engineer comes into play
- Example: choosing weighted sum vs. polynomial vs. neural network

Stage 2: Model Training

- Estimate the weights from training data
- Iteratively adjust parameters to minimize error
- This process of iteratively tuning weights is called **training** or **learning**

Stage 3: Inferencing

- Deploy the trained model with optimal parameters
- Process arbitrary real-life inputs not seen during training
- Generate outputs/predictions

Two-Step Model Estimation Process

1. **Model architecture selection**: Designing a parameterized function that we expect is a good proxy or surrogate for the unknown ideal function

2. **Training**: Estimating the parameters of that chosen function such that the outputs on training inputs match corresponding outputs as closely as possible

---

4. Data Preprocessing: Normalization

Why Normalize?

Input features often have different scales (e.g., income in thousands, age in years). Normalization transforms all features to a common scale.

Normalization Formula (Equation 1.1)

$$v_{\text{norm}} = \frac{v - v_{\text{min}}}{v_{\text{max}} - v_{\text{min}}}$$

This maps values from the input domain [v_min, v_max] to the range [0, 1].

Example:

- If hardness ranges from 0 to 100 (Mohs scale)
- A value of 50 normalizes to: (50 - 0)/(100 - 0) = 0.5

Vector Representation

A single input instance is represented as:

$$\vec{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \in [0,1]^2$$

---

5. Model Training: Loss Functions and Optimization

The Training Objective

We need to estimate the function that transforms the input vector to the output. Given training data with N+1 pairs:

$$\left(\vec{x}^{(0)}, y_{gt}^{(0)}\right), \left(\vec{x}^{(1)}, y_{gt}^{(1)}\right), \ldots, \left(\vec{x}^{(N)}, y_{gt}^{(N)}\right)$$

where subscript *gt* denotes **ground truth** (the known correct output).

Squared Error Loss

For the i-th training instance, the predicted output is:

$$y_{\text{predicted}}^{(i)} = \vec{w}^T \vec{x}^{(i)} + b$$

The **squared error** (loss) for this instance:

$$e_i^2 = \left(y_{\text{predicted}}^{(i)} - y_{gt}^{(i)}\right)^2$$

> ***Why square the error?*** *Squaring makes the error sign-independent. An error of +5 or -5 should be treated equally—both indicate the prediction is off by 5 units.*

Total Loss Function

The overall loss on the entire training data set is the sum of individual losses:

$$E^2 = \sum_{i=0}^{N} e_i^2 = \sum_{i=0}^{N} \left(y_{\text{predicted}}^{(i)} - y_{gt}^{(i)}\right)^2 = \sum_{i=0}^{N} \left(\vec{w}^T \vec{x}^{(i)} + b - y_{gt}^{(i)}\right)^2$$

Goal: Find weights $\vec{w}$ and bias b that minimize total error E^2.

Iterative Training Algorithm

In most cases, it is not possible to find a **closed-form solution** for optimal parameters. Instead, we use an **iterative approach**:

```
Algorithm 1.1: Training a Supervised Model
```

```
1. Initialize parameters $\vec{w}$, b with random values

2. WHILE $E^2$ > threshold DO:
   FOR each training instance $i \in [0, N]$:
       Adjust $\vec{w}$, b so that $E^2$ is reduced
   END FOR
END WHILE

3. Store final parameters as optimal: $\vec{w}_\star \leftarrow \vec{w}$, $b_\star \leftarrow b$
```

Key difference from classical mathematics:

- Mathematicians use closed-form solutions (all data at once)
- ML uses iterative solutions (process small portions at a time)
- This allows training on millions of items without holding all data in memory

---

6. Geometrical View of Machine Learning

Feature Space

Each input is a point in a high-dimensional space called the **feature space**—a space where all characteristic features examined by the model are represented.

- In our cat brain example: 2D space with axes X_0 (hardness) and X_1 (sharpness)
- In real problems: hundreds or thousands of dimensions
- Individual points denoted by coordinates (x_0, x_1) in lowercase

Output Space

The output y is a point in another space (typically lower-dimensional than feature space).

Key observation: The number of output dimensions is usually much smaller than the number of input dimensions.

The Geometric Interpretation

Geometrically speaking, a machine learning model maps a point in the feature space to a point in the output space.

The classification or estimation task is designed to be **easier in the output space** than in the feature space. Specifically:

> ***For classification:*** *Input points belonging to separate classes are expected to map to separate clusters in output space.*

Example: Cat Brain Geometry (Figure 1.2)

In the 2D feature space, the threat score can be modeled as the **signed distance** from the separator line $x_0 + x_1 = 1$:

$$y(x_0, x_1) = \frac{x_0 + x_1 - 1}{\sqrt{2}}$$

- **Positive y** → point is above the line → high threat → run away
- **y ≈ 0** → point is near the line → neutral → ignore
- **Negative y** → point is below the line → approach and purr

The optimal parameters are:

$$w_0 = \frac{1}{\sqrt{2}}, \quad w_1 = \frac{1}{\sqrt{2}}, \quad b = -\frac{1}{\sqrt{2}}$$

Higher-Dimensional Example (Figure 1.3)

A model maps points from input (feature) space to an output space where it is easier to separate classes:

- Input: Feature points belonging to two classes distributed over the volume of a cylinder in 3D
- Transformation: The model "unfurls" the cylinder into a rectangle
- Output: 2D planar space where classes can be discriminated with a simple linear separator

7. Linear vs. Nonlinear Models

The Limitation of Linear Separators

In Figure 1.2, classes could be separated by a line (hyperplane in higher dimensions). **This does not happen often in real life.**

When Linear Models Fail (Figure 1.4)

When classes are distributed such that no line can separate them, we need a **curved separator**:

- In 2D: curved line instead of straight line
- In 3D: curved surface instead of plane
- In higher dimensions: curved hypersurface instead of hyperplane

This requires **nonlinear model architectures**.

The Sigmoid Function

A very popular nonlinear function in machine learning is the **sigmoid function** (σ), so named because it looks like the letter S.

Definition (Equation 1.5):

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Properties:

- Output range: (0, 1)
- Smooth, differentiable everywhere
- As $x \to +\infty$, $\sigma(x) \to 1$
- As $x \to -\infty$, $\sigma(x) \to 0$
- $\sigma(0) = 0.5$

Nonlinear Model Architecture (Equation 1.6)

Applying sigmoid to the weighted sum:

$$y = \sigma\left(\vec{w}^T \vec{x} + b\right)$$

Key insight: The sigmoid imparts the nonlinearity. This architecture can handle relatively more complex classification tasks than the weighted sum alone.

> ***Equation 1.6 depicts the basic building block of a neural network.***

---

8. Deep Neural Networks: Achieving Higher Expressive Power

The Need for Expressive Power

In machine learning parlance, **the nonlinear model has more expressive power**—it can represent more complex decision boundaries.

Consider building a dog recognizer:

- Input: pixel locations and colors (x, y, r, g, b)
- Input dimensionality: proportional to number of pixels (very large)
- Massive variation in backgrounds, poses, lighting, breeds

We need a machine with really high expressive power. How do we create such a machine in a principled way?

The Cascaded Approach

Instead of generating output from input in a single step:

1. Generate a set of **intermediate (hidden) outputs** from inputs
2. Each hidden output is essentially a single logistic regression unit
3. Add another layer that takes previous layer output as input
4. Continue stacking layers
5. Combine outermost hidden layer outputs into final output

Multilayered Neural Network Architecture (Figure 1.7)

Notation conventions:

- Superscript identifies the layer (layer 0 closest to input, layer L furthest)

- Subscripts are two-dimensional: first identifies destination node, second identifies source node
- Weights for each layer form a matrix

Bias simplification: One of the inputs is set to x₀ = 1, and the corresponding weight (w₀) serves as the bias.

Layer Equations

Layer 0: Generates n₀ hidden outputs from n+1 inputs (Equation 1.7)

$$h_j^{(0)} = \sigma\left(\sum_{k=0}^{n} w_{jk}^{(0)} x_k\right) \quad \text{for } j = 0, 1, \ldots, n_0$$

Layer 1: Generates n₁ hidden outputs from n₀ hidden outputs (Equation 1.8)

$$h_j^{(1)} = \sigma\left(\sum_{k=0}^{n_0} w_{jk}^{(1)} h_k^{(0)}\right) \quad \text{for } j = 0, 1, \ldots, n_1$$

Final Layer (L): Generates m+1 visible outputs from n_{L-1} previous layer hidden outputs (Equation 1.9)

$$h_j^{(L)} = \sigma\left(\sum_{k=0}^{n_{L-1}} w_{jk}^{(L)} h_k^{(L-1)}\right) \quad \text{for } j = 0, 1, \ldots, m$$

The Power of Depth

The machine depicted in Figure 1.7 can be **incredibly powerful, with huge expressive power**. We can adjust its expressive power systematically by:

- Adding more layers (depth)
- Adding more nodes per layer (width)
- Choosing appropriate activation functions

This is a neural network. Deep learning is nothing but a **multilayered nonlinear machine**.

Summary: Key Mental Pictures

1. **Paradigm shift:** In traditional computing, we program explicit rules. In ML, we build mathematical models that **approximate unknown functions** from data.
2. **Function approximation view:** ML models are parameterized proxies for unknown classification/estimation functions. Parameters are estimated via training.
3. **Training process:** Iteratively adjust weights to minimize the difference between predicted and target outputs on training data.
4. **Geometric view:** A model is a **transformation** mapping points from high-dimensional feature space to lower-dimensional output space where the task becomes simpler.
5. **Expressive power:** More complex tasks require models with greater expressive power. This comes from:
6. **Nonlinearity** (e.g., sigmoid function)
7. **Layered combination** of simpler machines
8. **Deep learning = multilayered nonlinear machine:** Complex model functions are built by combining simpler basis functions through multiple layers.

Prepared from: "Math and Architectures of Deep Learning" - Chapter 1